Lecture 2
(part1)

Topics covered:
Instruction Set Architecture

Execution of an instruction

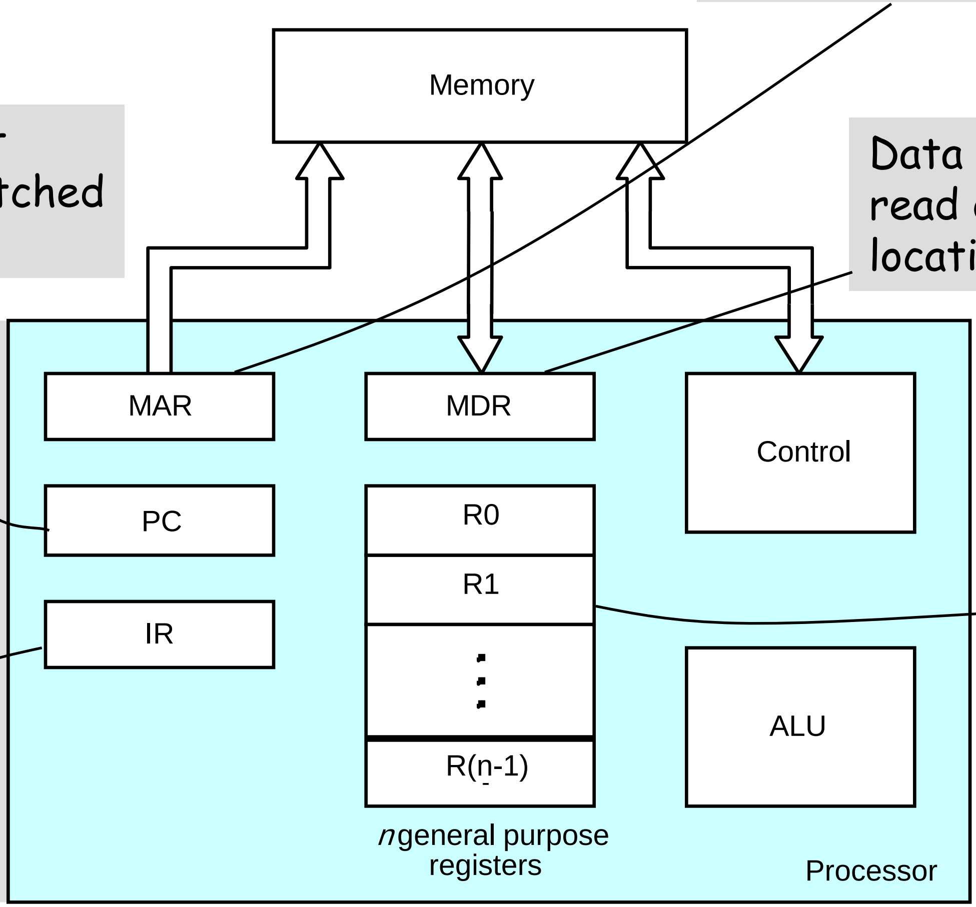
- Recall the steps involved in the execution of an instruction by a processor:
  - Fetch an instruction from the memory.
  - Fetch the operands.
  - Execute the instruction.
  - Store the results.
- Several issues:
  - Where is the address of the memory location from which the present instruction is to be fetched?
  - Where is the present instruction stored while it is executed?
  - Where and what is the address of the memory location from which the data is fetched?
  - ......
- Basic processor architecture has several registers to assist in the execution of the instructions.

Basic processor architecture

Address of the memory location to be accessed

Address of the next instruction to be fetched and executed.

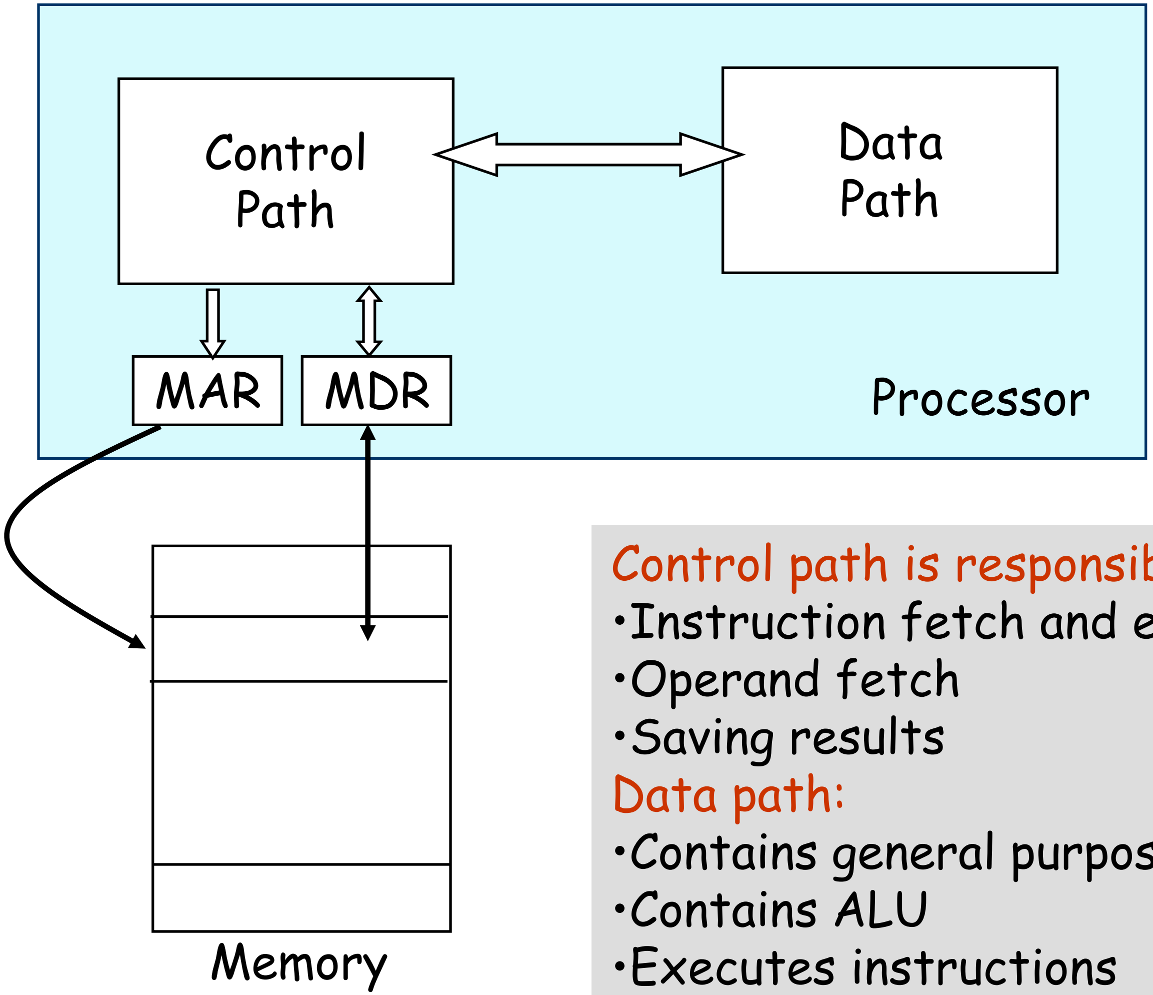
Data to be read into or read out of the current location



General purpose registers

Instruction that is currently being executed

Basic processor architecture (contd..)



Control path is responsible for:

- Instruction fetch and execution sequencing
- Operand fetch
- Saving results

Data path:

- Contains general purpose registers
- Contains ALU
- Executes instructions

Registers in the control path

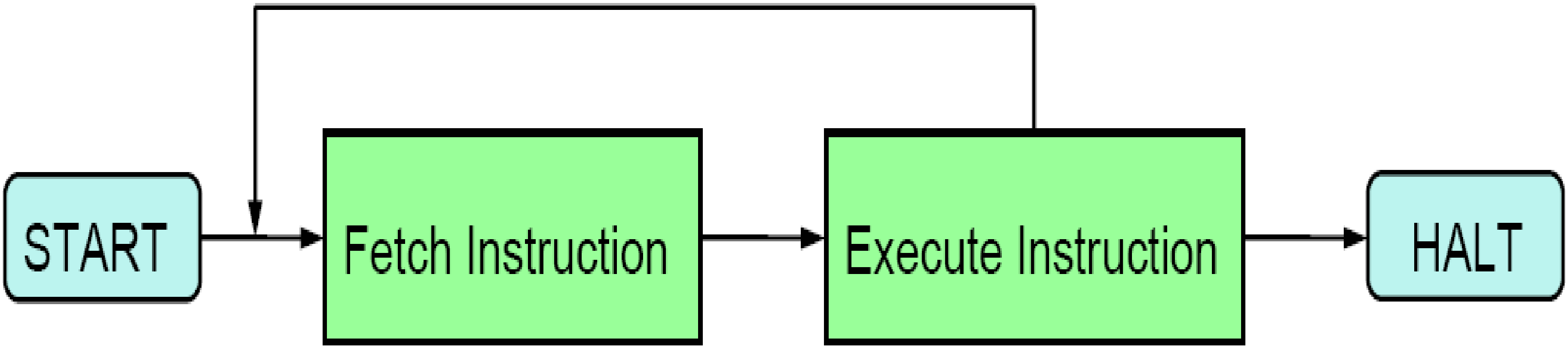
- **Instruction Register (IR):**
  - Instruction that is currently being executed.
- **Program Counter (PC):**
  - Address of the next instruction to be fetched and executed.
- **Memory Address Register (MAR):**
  - Address of the memory location to be accessed.
- **Memory Data Register (MDR):**
  - Data to be read into or read out of the current memory location, whose address is in the Memory Address Register (MAR).

Fetch/Execute cycle

- Execution of an instruction takes place in two phases:
  - Instruction fetch.
  - Instruction execute.
- Instruction fetch:
  - Fetch the instruction from the memory location whose address is in the Program Counter (PC).
  - Place the instruction in the Instruction Register (IR).
- Instruction execute:
  - Instruction in the IR is examined (decoded) to determine which operation is to be performed.
  - Fetch the operands from the memory or registers.
  - Execute the operation.
  - Store the results in the destination location.
- Basic fetch/execute cycle repeats indefinitely.

Instruction Execution

- Basic instruction cycle



Memory organization

- ❑ Recall:
  - ◆ Information is stored in the memory as a collection of bits.
  - ◆ Collection of bits are stored or retrieved simultaneously is called a word.
  - ◆ Number of bits in a word is called word length.
  - ◆ Word length can be 16 to 64 bits.
- ❑ Another collection which is more basic than a word:
  - ◆ Collection of 8 bits known as a "byte"
- ❑ Bytes are grouped into words, word length can also be expressed as a number of bytes instead of the number of bits:
  - ◆ Word length of 16 bits, is equivalent to word length of 2 bytes.
- ❑ Words may be 2 bytes (older architectures), 4 bytes (current architectures), or 8+ bytes (modern architectures).

Memory organization (contd..)

- Accessing the memory to obtain information requires specifying the "address" of the memory location.
- Recall that a memory has a sequence of bits:
  - Assigning addresses to each bit is impractical and unnecessary.
  - Typically, addresses are assigned to a single byte.
  - "Byte addressable memory"
- Suppose k bits are used to hold the address of a memory location:

Size of the memory in bytes is given by: 2^k
where k is the number of bits used to hold a memory address.
E.g., for a 16-bit address, size of the memory is 2^{16}= 65536 bytes

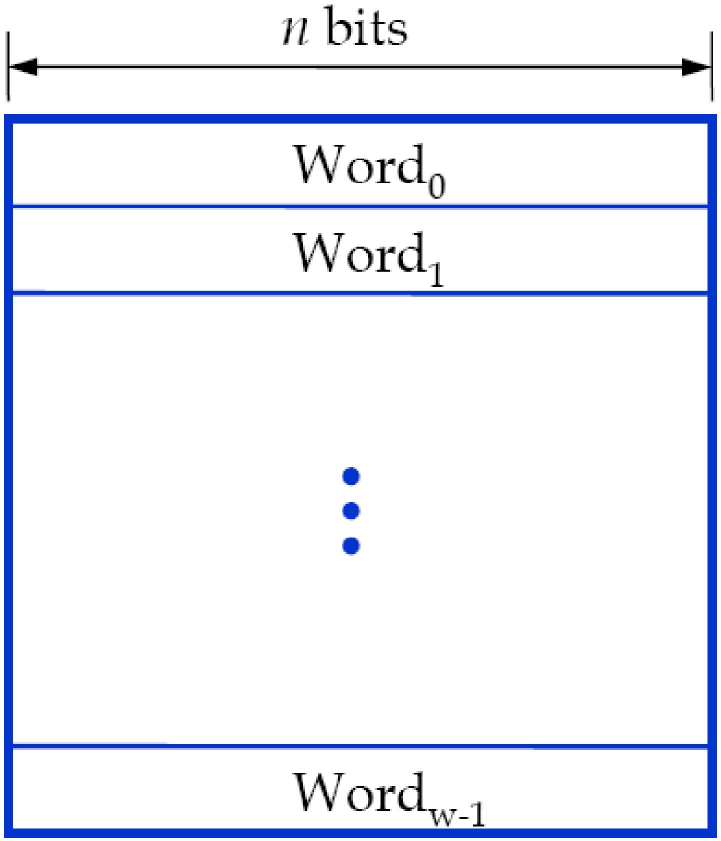
What is the size of the memory for a 24-bit address?

Memory Addresses

- For example,
  a 24-bit address generates an address space of 2^{24} (16,777,216) locations
- Terminology
  - 2^{10}: 1K (kilo)
  - 2^{20}: 1M (mega)
  - 2^{30}: 1G (giga)
  - 2^{40}: 1T (tera)

Memory Words

Memory words

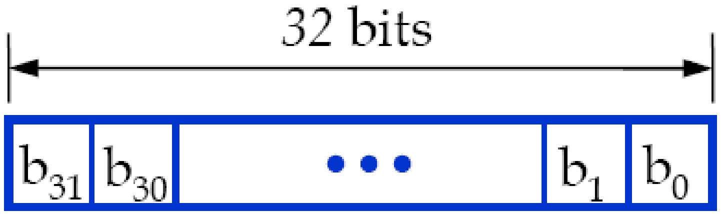

n bits

Word$_0$

Word$_1$

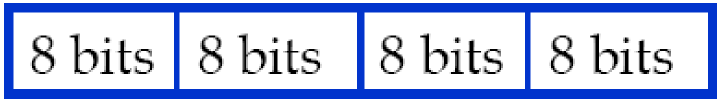
Word$_{w-1}$

A signed integer


32 bits

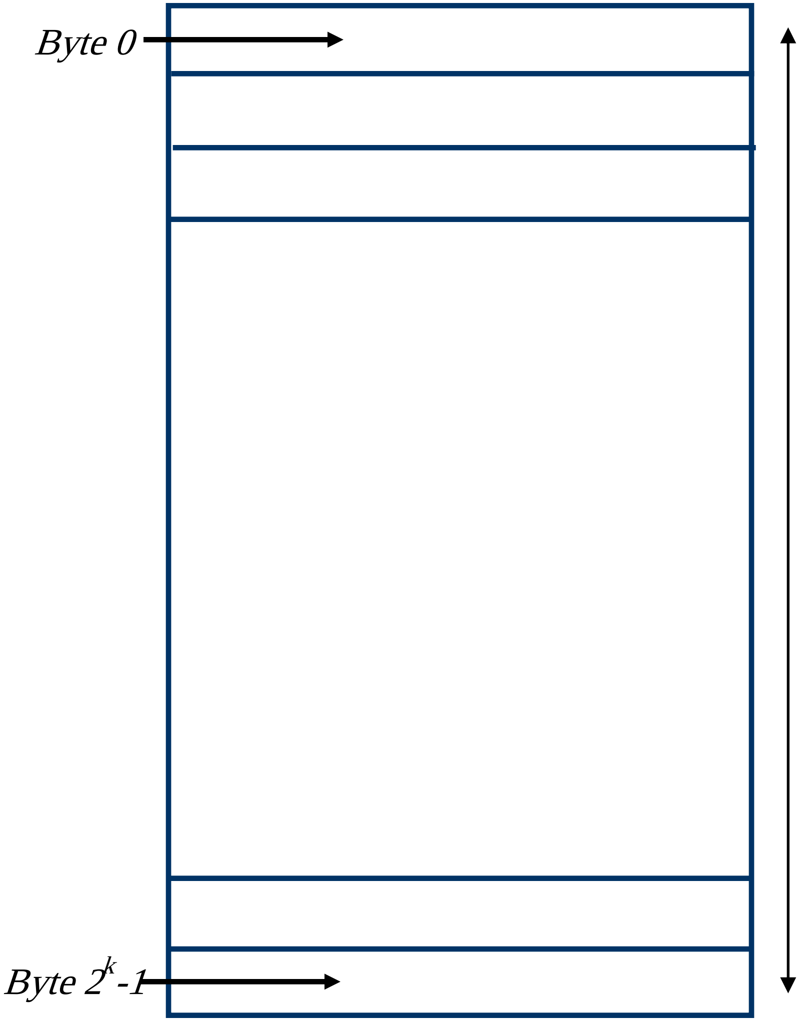
b_{31} b_{30} • • • b_1 b_0

Four characters


8 bits | 8 bits | 8 bits | 8 bits

ASCII
character

Memory organization (contd..)



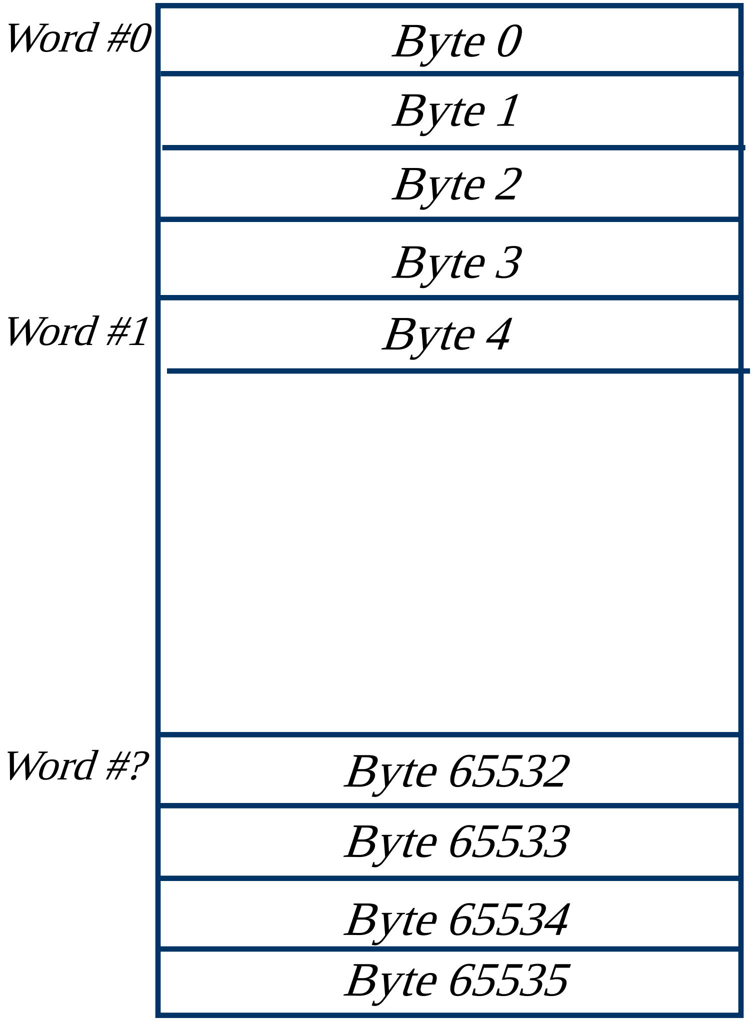
- Memory is viewed as a sequence of bytes.
- Address of the first byte is 0
- Address of the last byte is 2^k - 1, where k is the number of bits used to hold memory address
- E.g. when k = 16,

Address of the first byte is 0
Address of the last byte is 65535

- E.g. when k = 2,

Address of the first byte is ?
Address of the last byte is ?

Memory organization (contd..)



Word #0

Byte 0

Byte 1

Byte 2

Byte 3

Word #1

Byte 4

Word #?

Byte 65532

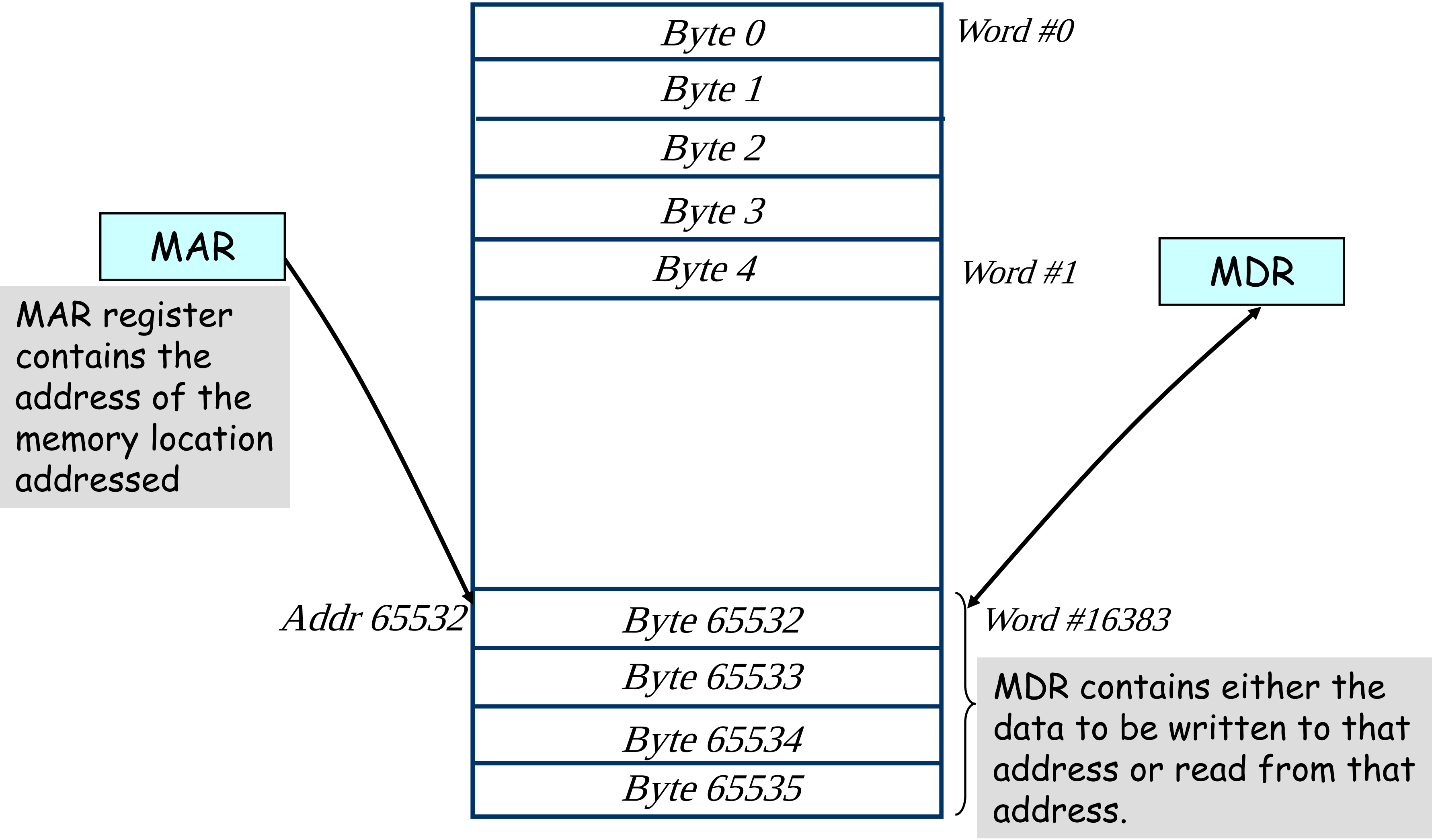
Byte 65533

Byte 65534

Byte 65535

Consider a memory organization:
16-bit memory addresses
Size of the memory is ?
Word length is 4 bytes
Number of words = $\dfrac{\text{Memory size(bytes)}}{\text{Word length(bytes)}}$ = ?
Word #0 starts at Byte #0.
Word #1 starts at Byte #4.
Last word (Word #?) starts at Byte#?

Memory organization (contd..)



Byte 0

Byte 1

Byte 2

Byte 3

Byte 4

Byte 65532

Byte 65533

Byte 65534

Byte 65535

Word #0

Word #1

Word #16383

Addr 65532

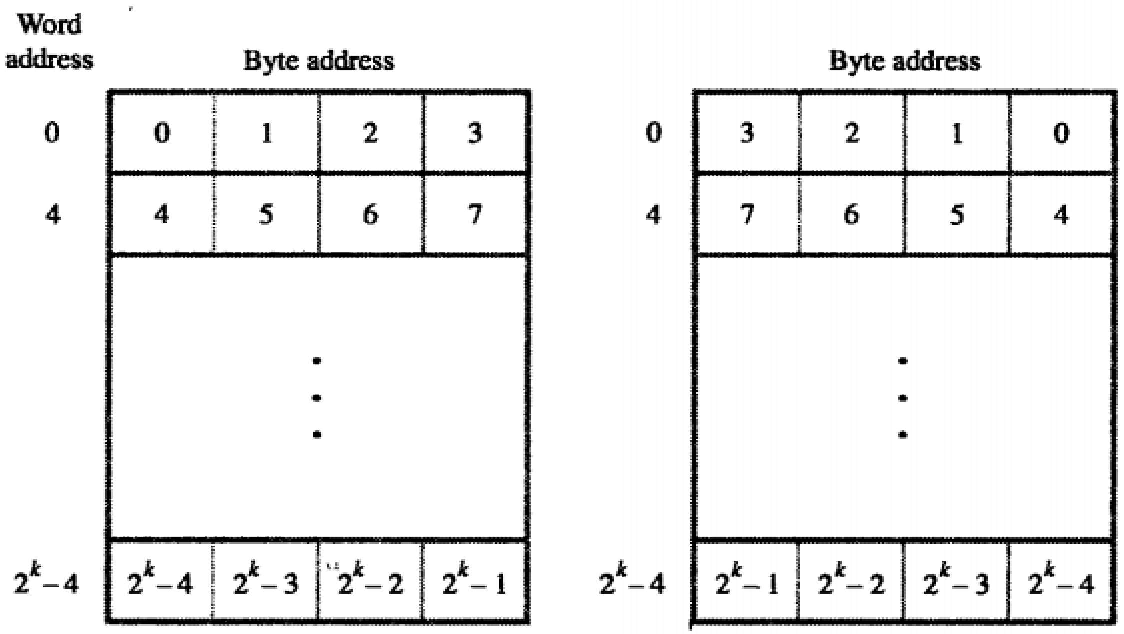
MAR

MAR register contains the address of the memory location addressed

MDR

MDR contains either the data to be written to that address or read from that address.

Byte addresses assignment



Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
		⋮		
2^k-4	2^k-4	2^k-3	2^k-2	2^k-1

(a) Big-endian assignment

	Byte address			
0	3	2	1	0
4	7	6	5	4
		⋮		
2^k-4	2^k-1	2^k-2	2^k-3	2^k-4

(b) Little-endian assignment

- <u>Big-endian assignment</u>: Lower byte addresses are used for the most significant bytes.
- <u>Little-endian assignment</u>: Lower byte addresses are used for the least significant bytes.

Memory operations

- Memory read or load:
  - Place address of the memory location to be read from into MAR.
  - Issue a Memory_read command to the memory.
  - Data read from the memory is placed into MDR automatically (by control logic).
- Memory write or store:
  - Place address of the memory location to be written to into MAR.
  - Place data to be written into MDR.
  - Issue Memory_write command to the memory.
  - Data in MDR is written to the memory automatically (by control logic).

Instruction types

- Computer instructions must be capable of performing 4 types of operations.
- Data transfer/movement between memory and processor registers.
  - E.g., memory read, memory write
- Arithmetic and logic operations:
  - E.g., addition, subtraction, comparison between two numbers.
- Program sequencing and flow of control:
  - Branch instructions
- Input/output transfers to transfer data to and from the real world.

Instruction types (contd..)

- Examples of different types of instructions in assembly language notation.
- Data transfers between processor and memory.
  - *Move A, B* (B = A).
  - *Move A, R1* (R1 = A).
- Arithmetic and logic operation:
  - *Add A, B, C* (C = A + B)
- Sequencing(Branching):
  - *Jump Label* (Jump to the subroutine which starts at Label).
- Input/output data transfer:
  - *Input PORT, R5* (Read from i/o port "PORT" to register R5).
  - *Output R1, PORT.* (write into i/o port "PORT" from register R1).

Instructions

- Register transfer notation
  - The contents of a location are denoted by placing square brackets around the name of the location
  - For example, R1←[LOC] means that the contents of memory location LOC are transferred into processor register R1
  - As another example, R3←[R1]+[R2] means that adds the contents of registers R1 and R2, and then places their sum into register R3

Specifying operands in instructions

- Operands are the data operated upon by the instructions.
- Recall that operands may have to be fetched from a memory location to execute an operation.
  - Memory locations have addresses using which they can be accessed.
- Operands may also be stored in the general purpose registers.
  - Intermediate value of some computation which will be required immediately for subsequent computations.
  - Registers also have addresses.
- Specifying the operands on which the instruction is to operate involves specifying the addresses of the operands.
  - ***Address can be of a memory location or a register.***

Source and destination operands

- Operation may be specified as:
  - *Operation source1, source2, destination*
- An operand is called a source operand if:
  - It appears on the right-hand side of an expression
    - E.g., *Add A, B, C (C = A+ B)*
      - A and B are source operands.
- An operand is called a destination operand if:
  - It appears on the left-hand side of an expression.
    - E.g., *Add A, B, C (C = A + B)*
      - C is a destination operand.

Source and destination operands (contd..)

- In case of some instructions, the same operand serves as both the source and the destination.
  - Same operand appears on the right and left side of an expression.
    - E.g. *Add A, B (B = A + B)*
    - B is both the source and the destination operand.
- Another classification of instructions is based on the number of operand addresses in the instruction.

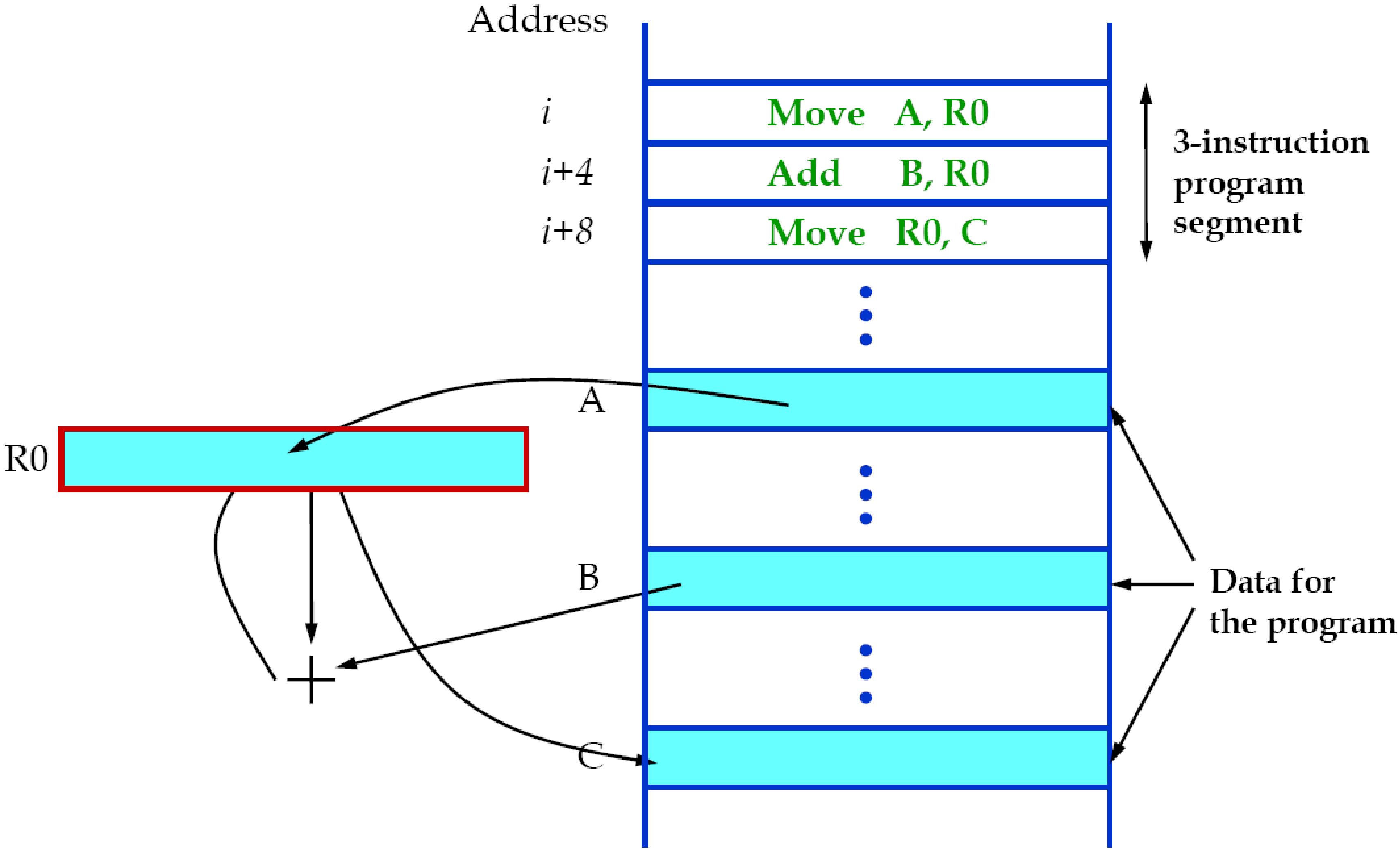
Instruction types

- Instructions can also be classified based on the number of operand addresses they include.
  - 3, 2, 1, 0 operand addresses.
- 3-address instructions are almost always instructions that implement binary operations.
  - E.g. *Add A, B, C (C = A + B)*
  - k bits are used to specify the address of a memory location, then 3-address instructions need 3*k bits to specify the operand addresses.
  - 3-address instructions, where operand addresses are memory locations are too big to fit in one word.

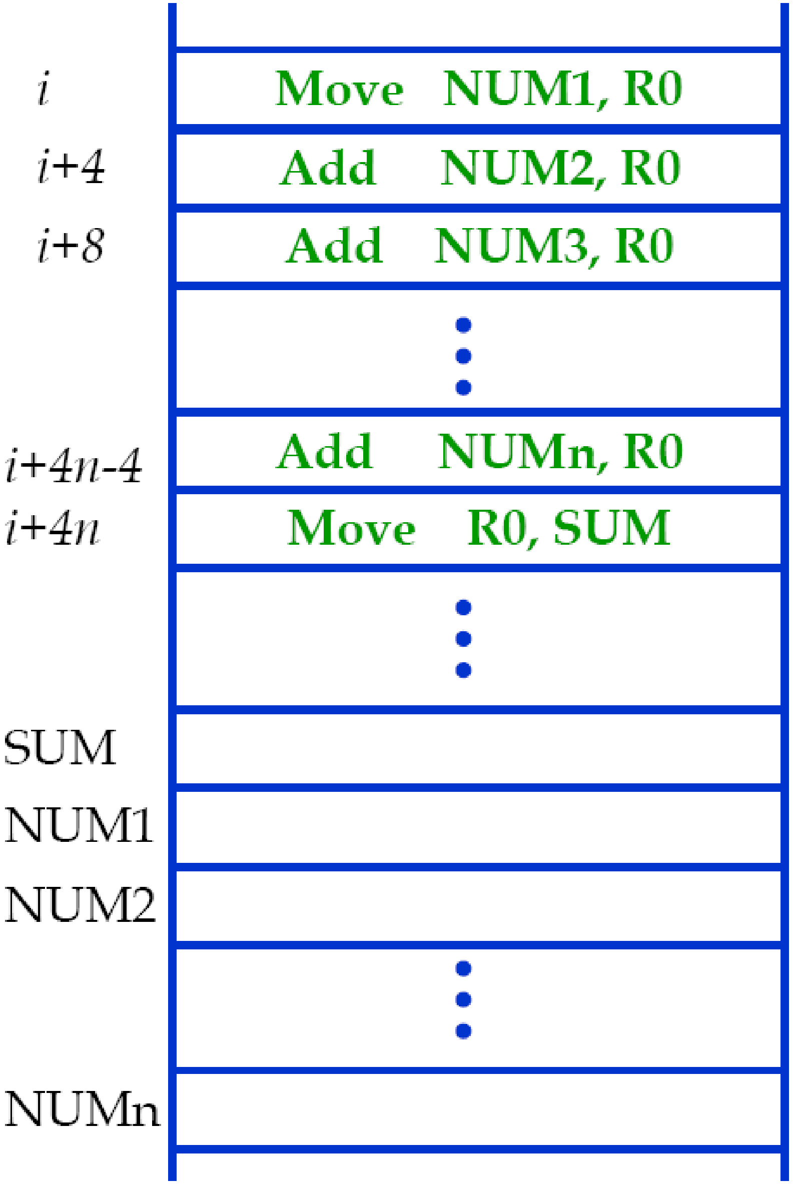
Instruction types (contd..)

- 2-address instructions one operand serves as a source and destination:
  - E.g. *Add A, B (B = A + B)*
  - 2-address instructions need 2*k bits to specify an instruction.
  - This may also be too big to fit into a word.
- 2-address instructions, where at least one operand is a processor register:
  - E.g. *Add A, R1 (R1 = A + R1)*
- 1-address instructions require only one operand.
  - E.g. *Clear A (A = 0)*
- 0-address instructions do not operate on operands.
  - E.g. *Halt (Halt the computer)*
- ***How are addresses of operands specified in the instructions?***

A Program for C←-[A]+[B]

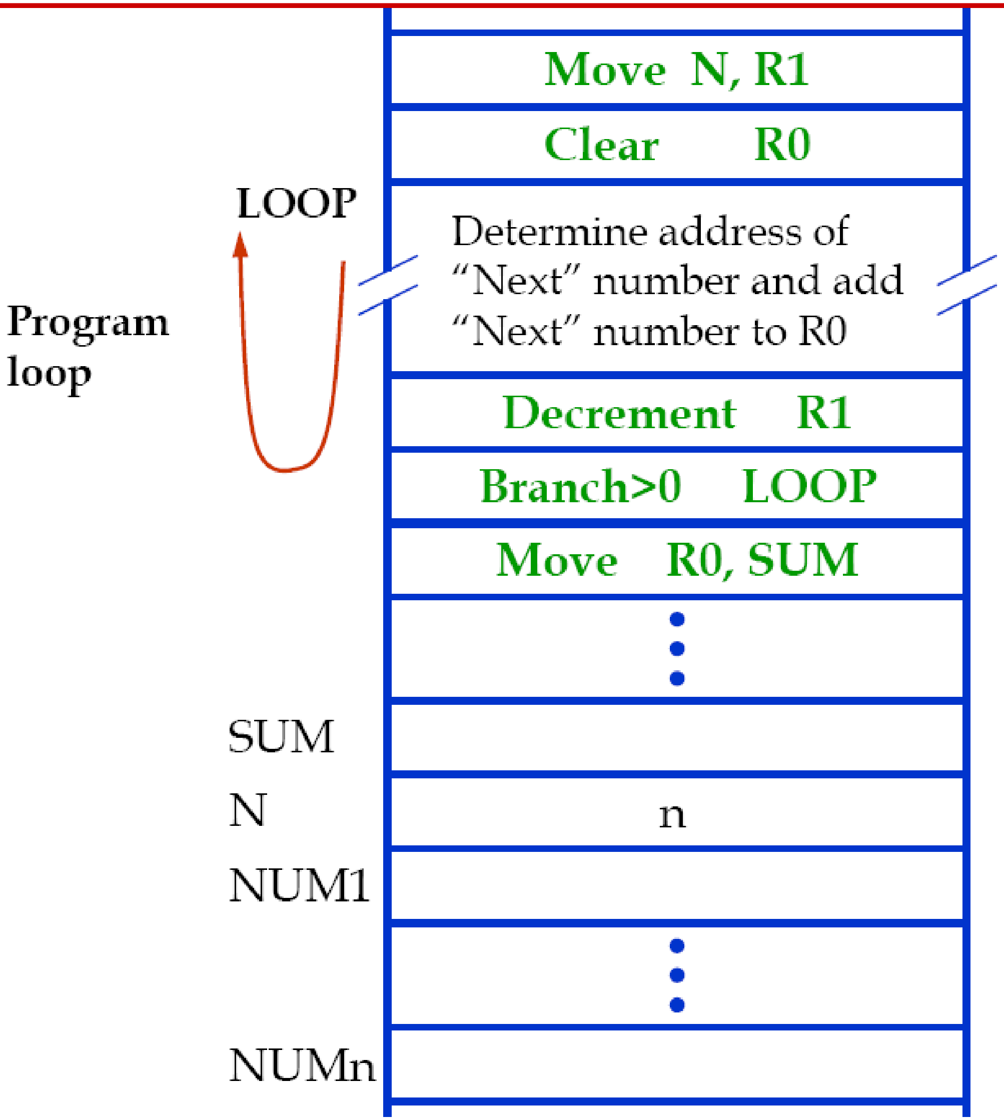


Straight-Line Sequencing



i	Move NUM1, R0
i+4	Add NUM2, R0
i+8	Add NUM3, R0
	⋮
i+4*n*-4	Add NUMn, R0
i+4*n*	Move R0, SUM
	⋮
SUM	
NUM1	
NUM2	
	⋮
NUMn	

Branching



Move N, R1

Clear R0

Determine address of "Next" number and add "Next" number to R0

Decrement R1

Branch>0 LOOP

Move R0, SUM

n

Condition Codes

- The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions.
- This is accomplished by recording required information in individual bits, often called *condition code flags*

Condition Codes

- Four commonly used flags are
  - N (negative): set to 1 if the results is negative; otherwise, cleared to 0
  - Z (zero): set to 1 if the result is 0; otherwise, cleared to 0
  - V (overflow): set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
  - C (carry): set to 1 if a carry-out results from the operation otherwise, cleared to 0
- N and Z flags caused by an arithmetic or a logic operation,
- V and C flags caused by an arithmetic operation

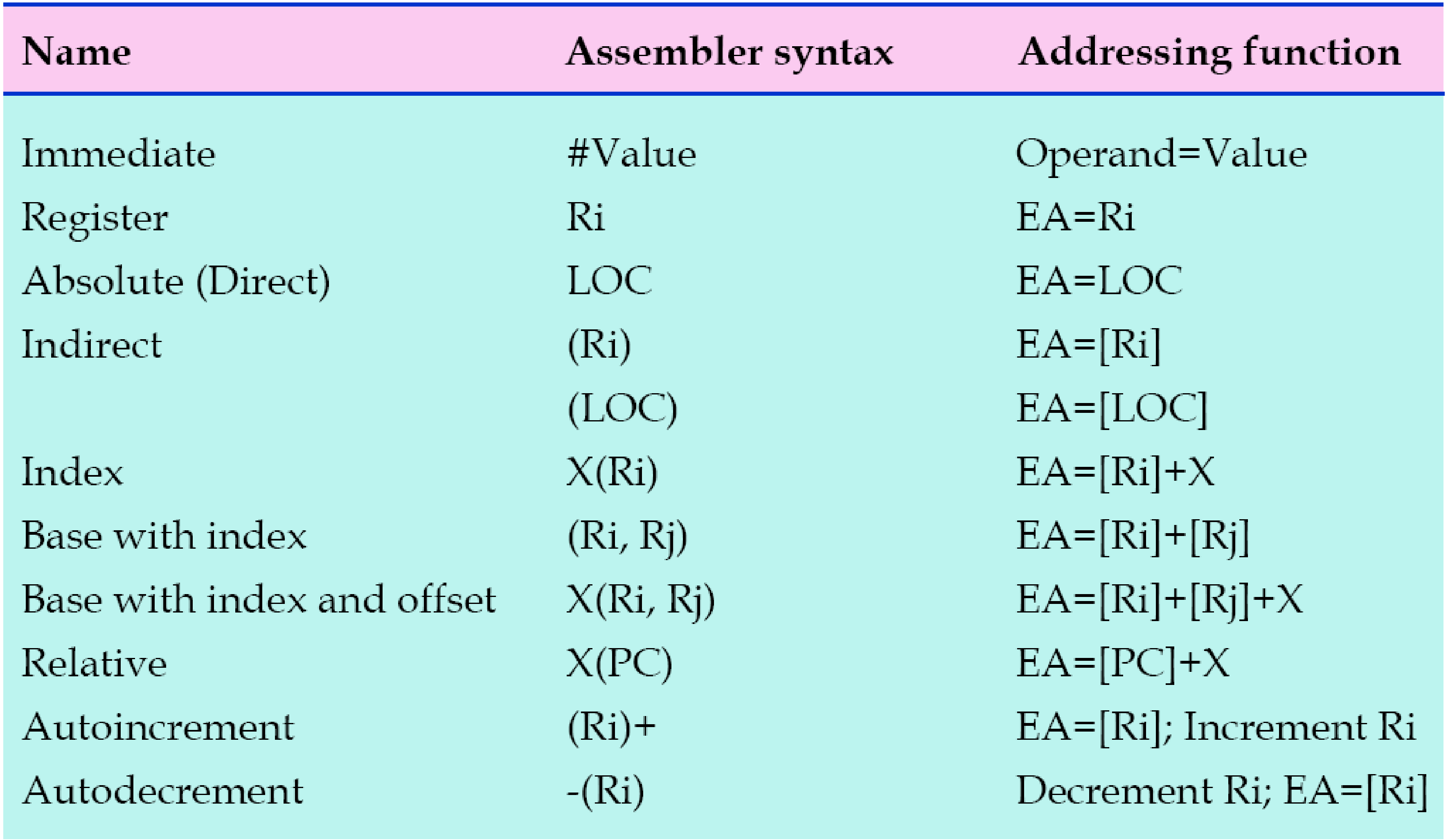
Addressing Modes

- A high-level language enables the programmer to use constants, local and global variables, pointers, and arrays

- When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities in the instruction set of the computer

- ***The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes***

Generic Addressing Modes



Name	Assembler syntax	Addressing function
Immediate	#Value	Operand=Value
Register	Ri	EA=Ri
Absolute (Direct)	LOC	EA=LOC
Indirect	(Ri)	EA=[Ri]
	(LOC)	EA=[LOC]
Index	X(Ri)	EA=[Ri]+X
Base with index	(Ri, Rj)	EA=[Ri]+[Rj]
Base with index and offset	X(Ri, Rj)	EA=[Ri]+[Rj]+X
Relative	X(PC)	EA=[PC]+X
Autoincrement	(Ri)+	EA=[Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA=[Ri]

Addressing modes

- Different ways in which the address of an operand is specified in an instruction is referred to as addressing modes.
- Register mode
  - Operand is the contents of a processor register.
  - Address of the register (its Name) is given in the instruction.
  - E.g. *Clear R1* *or* *Move R1, R2*
- Absolute mode
  - Operand is in a memory location.
  - Address of the memory location is given explicitly in the instruction.
  - E.g. *Clear A* *or* *Move LOC, R2*
  - Also called as "Direct mode" in some assembly languages
- Register and absolute modes can be used to represent ***variables***

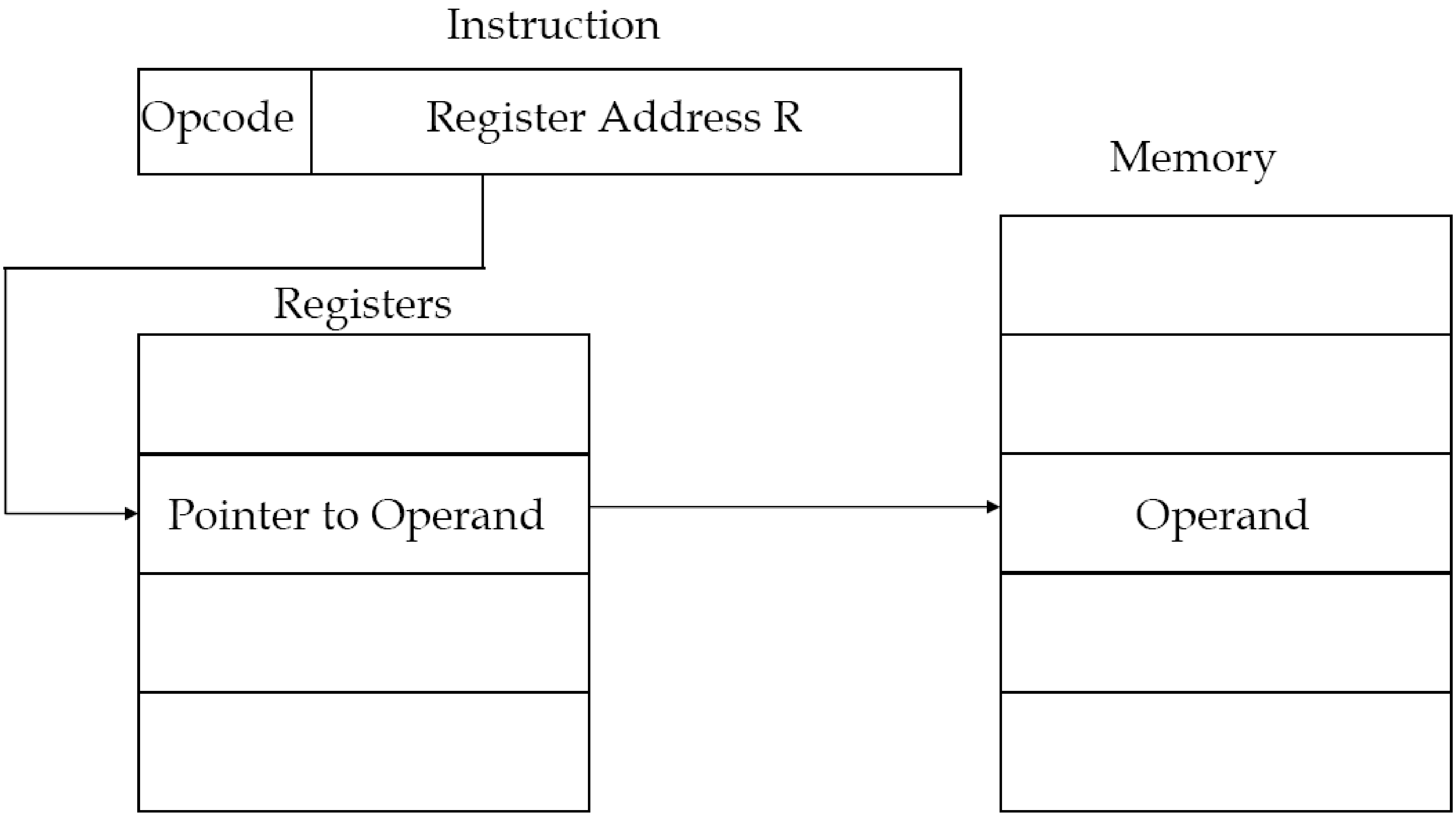
Addressing modes (contd..)

- Immediate mode
  - Operand is given explicitly in the instruction.
  - E.g. *Move #200, R0*
  - Can be used to represent ***constants***.
- Register, Absolute and Immediate modes contained either the address of the operand or the operand itself.
- Some instructions provide information from which the memory address of the operand can be determined
  - That is, they provide the "Effective Address" of the operand.
  - They do not provide the operand or the address of the operand explicitly.
- ***Different ways in which "Effective Address" of the operand can be generated.***

Indirection and Pointers

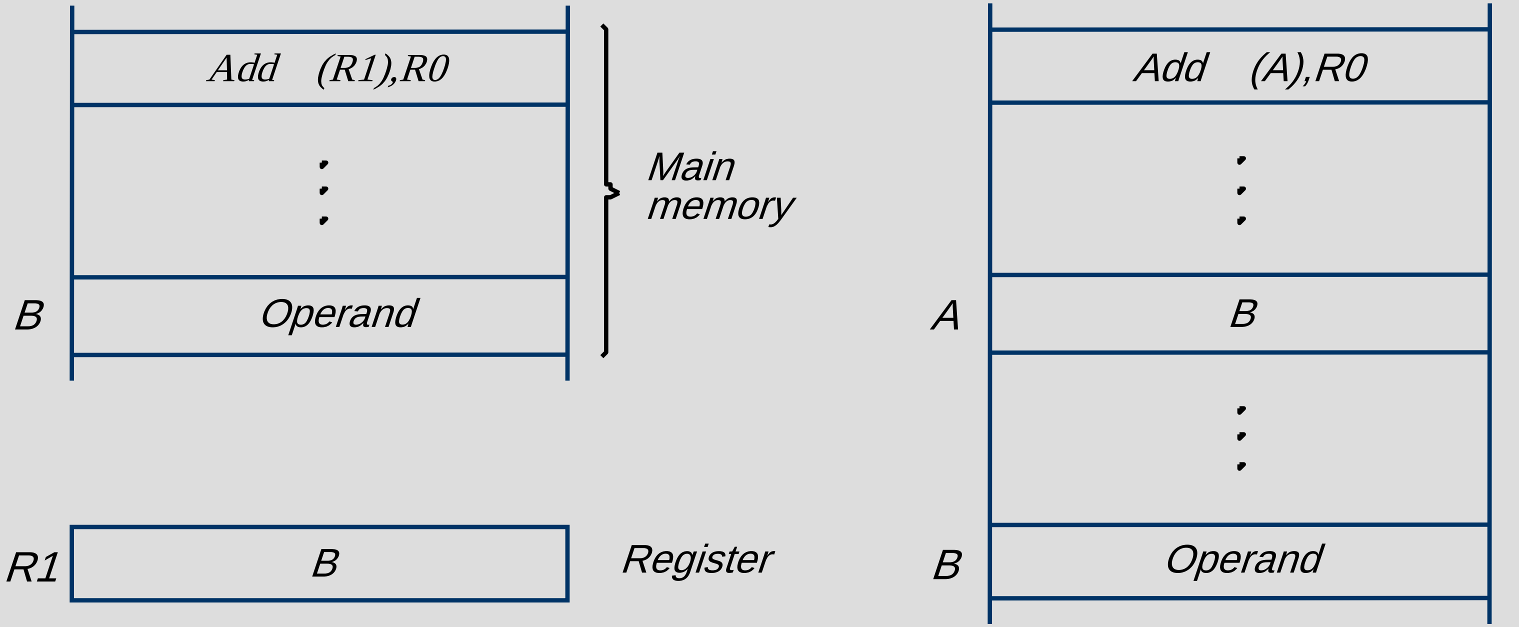
- Indirect mode: the effective address of the operand is the contents of a register or memory location whose address appears in the instruction
- Indirection is denoted by placing the name of the register or the memory address given in the instruction in parentheses
- The register or memory location that contains the address of an operand is called a pointer

Register Indirect Addressing Diagram



Addressing modes (contd..)

Effective Address of the operand is the contents of a register or a memory location whose address appears in the instruction.



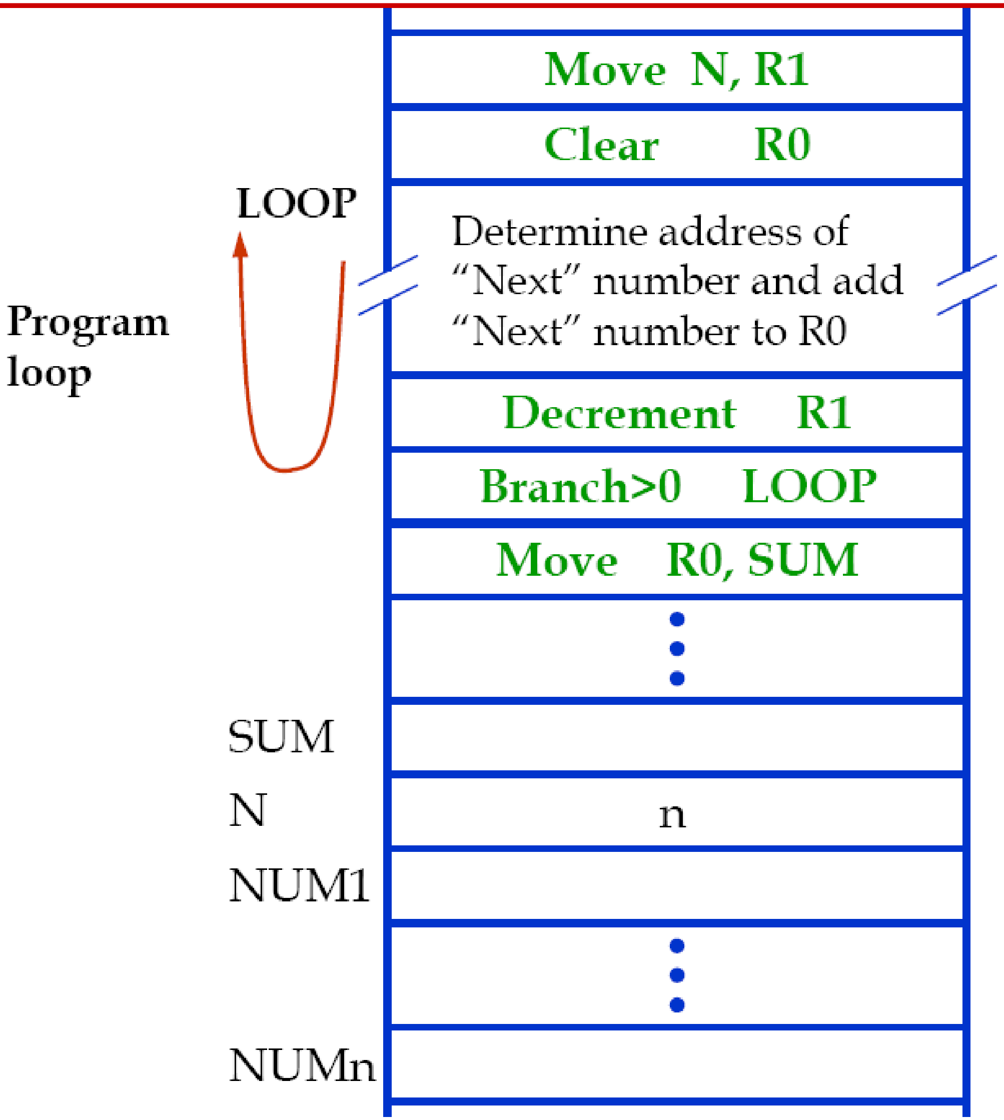
- Register R1 contains Address B
- Address B has the operand

- Address A contains Address B
- Address B has the operand

***R1* and *A* are called "*pointers*"**

This is called as "Indirect Mode"

Using Indirect Addressing in a Program



Move N, R1

Clear R0

LOOP

Determine address of "Next" number and add "Next" number to R0

Program loop

Decrement R1

Branch>0 LOOP

Move R0, SUM

SUM

N n

NUM1

NUMn

Using Indirect Addressing in a Program

Address	Contents		
	Move	N, R1	Initialization
	Move	#NUM1, R2	
	Clear	R0	
LOOP	Add	(R2), R0	
	Add	#4, R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0, SUM	

Indexing and Arrays

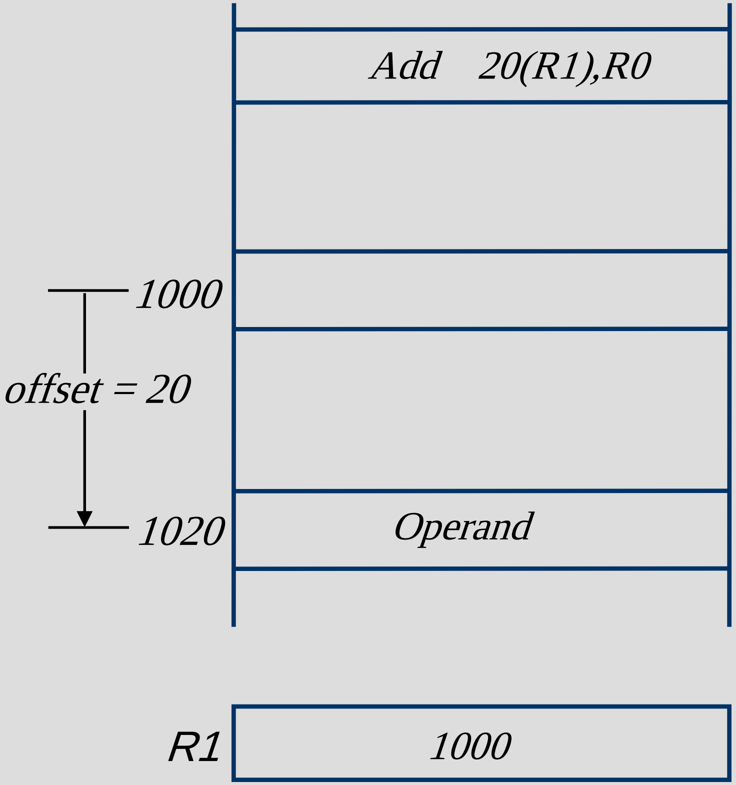
- Index mode: the effective address of the operand is generated by adding a constant value to the contents of a register
  - The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general purpose registers in the processor.
  - It is referred to as an *index register*

Indexing and Arrays

- The index mode is useful in dealing with lists and arrays
- We denote the Index mode symbolically as X(Ri), where X denotes the constant value contained in the instruction and Ri is the name of the register involved.
- The effective address of the operand is given by EA=X+(Ri).
- The contents of the index register are not changed in the process of generating the effective address

Addressing modes (contd..)

Effective Address of the operand is generated by adding a constant value to the contents of the register

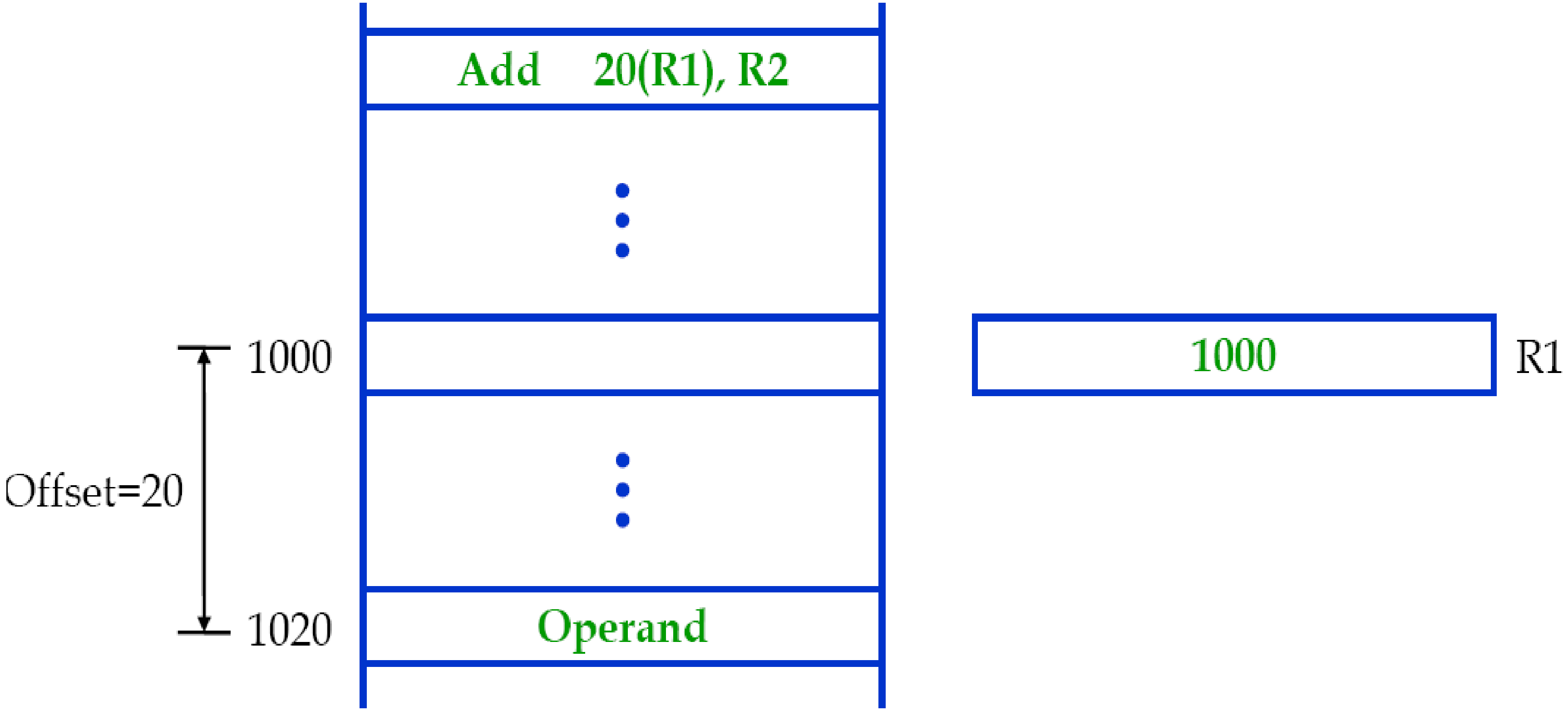


- *Operand is at address 1020*
- *Register R1 contains 1000*
- *Offset 20 is added to the contents of R1 to generate the address **1020***
- *Contents of R1 do not change in the process of generating the address*
- *R1 is called as an "index register"*

This is the "Indexing Mode"

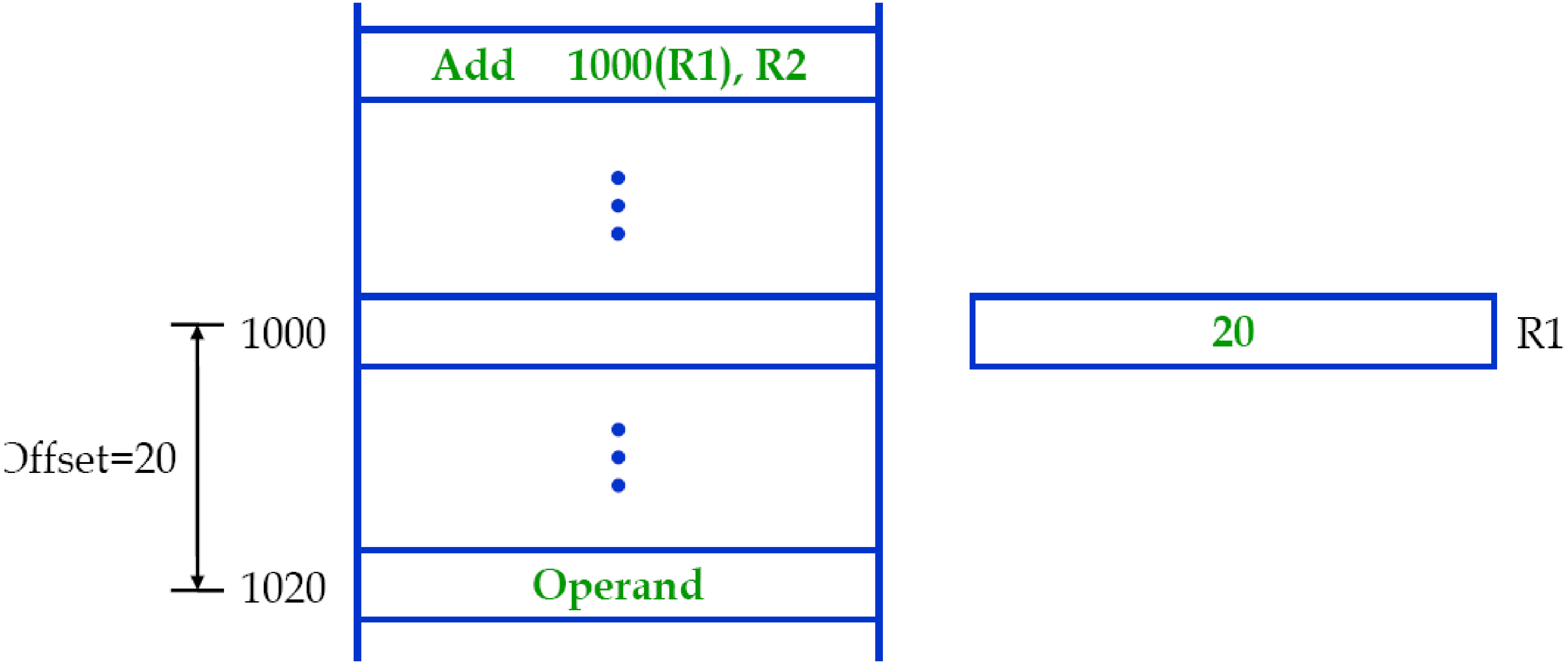
Indexed Addressing

Offset is given as a constant



Indexed Addressing

Offset is in the index register



Add 1000(R1), R2

1000

Offset=20

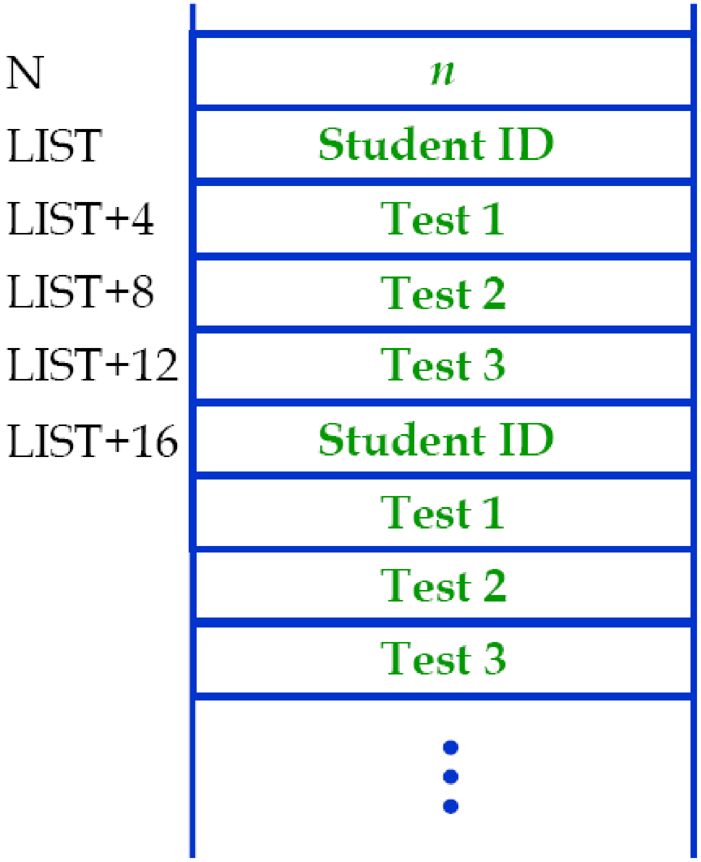
1020

Operand

20

R1

An Example for Indexed Addressing



N	*n*
LIST	Student ID
LIST+4	Test 1
LIST+8	Test 2
LIST+12	Test 3
LIST+16	Student ID
	Test 1
	Test 2
	Test 3
	⋮

```
        Move         #LIST, R0
        Clear        R1
        Clear        R2
        Clear        R3
        Move         N, R4
LOOP    Add          4(R0), R1
        Add          8(R0), R2
        Add          12(R0), R3
        Add          #16, R0
        Decrement    R4
        Branch>0     LOOP
        Move         R1, SUM1
        Move         R2, SUM2
        Move         R3, SUM3
```

Variations of Indexed Addressing Mode

- A second register may be used to contain the offset X, in which case we can write the Index mode as (Ri,Rj)
  - The effective address is the sum of the contents of registers Ri and Rj
  - The second register is usually called the base register
  - This mode implements a two-dimensional array
- Another version of the Index mode use two registers plus a constant, which can be denoted as X(Ri,Rj)
  - The effective address is the sum of the constant X and the contents of registers Ri and Rj
  - This mode implements a three-dimensional array

Addressing Modes (contd..) Relative mode

- Effective Address of the operand is generated by adding a constant value to the contents of the Program Counter (PC).
- Variation of the Indexing Mode, where the index register is the PC instead of a general purpose register.
- When the instruction is being executed, the PC holds the address of the next instruction in the program.
- Useful for specifying target addresses in branch instructions.
- Addressed location is "relative" to the PC, this is called "Relative Mode"

Addressing Modes (contd..) Relative mode

- The Instruction Branch > 0 Loop
- Suppose that the loop starts at address 1000, and the branch instruction at address 1012.
- The PC value now is 1016.
- To branch to location Loop (1000), the offset value is 1000 - 1016 = -16
- When the assembler processes such instruction, it computes the required offset value, and generates the corresponding machine instruction using the addressing mode:

-16(PC)

Addressing Modes (contd..)

- Autoincrement mode:
  - Effective address of the operand is the contents of a register specified in the instruction.
  - **After** accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location.
  - ***(R1)+***
- Autodecrement mode
  - The contents of a register specified in the instruction are decremented. Then, these contents are used as the effective address of the operand.
  - ***-(R1)***
- Autoincrement and Autodecrement modes are useful for implementing "Last-In-First-Out" data structures.

Addressing modes (contd..)

- Implicitly the increment and decrement amounts are 1.
  - This would allow us to access individual bytes in a byte addressable memory.
- Recall that the information is stored and retrieved one word at a time.
  - In most computers, increment and decrement amounts are equal to the word size in bytes.
- E.g., if the word size is 4 bytes (32 bits):
  - Autoincrement increments the contents by 4.
  - Autodecrement decrements the contents by 4.

An Example of Autoincrement Addressing

```
        Move        N, R1
        Move        #NUM1, R2
        Clear       R0
LOOP    Add         (R2)+, R0
        Decrement   R1
        Branch>0    LOOP
        Move        R0, SUM
```